Towards model checking executable UML specifications in mCRL2

Hansen, H.H.; Ketema, J.; Mousavi, M.R.; Luttik, B.; van de Pol, J.C.

published in
Innovations in Systems and Software Engineering
2011

DOI (link to publisher)
10.1007/s11334-009-0116-1

document version
Publisher's PDF, also known as Version of record

Innovations Syst Softw Eng manuscript No.
(will be inserted by the editor)

Towards Model Checking Executable UML Specifications in mCRL2

Helle Hvid Hansen · Jeroen Ketema · Bas Luttik · MohammadReza Mousavi · Jaco van de Pol

Abstract We describe a translation of a subset of Executable UML (xUML) into the process algebraic specification language mCRL2. This subset includes class diagrams with class generalisations, and state machines with send and change events. The choice of these xUML constructs is dictated by their use in the modelling of railway interlocking systems.

The long term goal is to verify safety properties of interlockings modelled in xUML using the mCRL2 and LTSmin toolsets. Initial verification of an interlocking toy example demonstrates that the safety properties of model instances depend crucially on the run-to-completion assumptions.

Keywords software verification and validation, specification languages, model checking, executable UML, process algebra

1 Introduction

We translate a subset of Executable UML (xUML) [15] into the formal specification language mCRL2 [12] with the purpose of verifying safety properties. The xUML constructs covered include class diagrams with class generalisations and object associations, and state machines which consist of composite and concurrent states and have signal and change events.

The mCRL2 language extends the process algebra ACP [3] with abstract data types. Its process algebraic foundation makes mCRL2 suitable for specifying dynamic, concurrent behaviour. Moreover, it enables the use of compositional verification methods and provides a clear, formal semantics [12]. Our use of mCRL2 is strongly motivated by the availability of powerful verification tools: The mCRL2 toolset[1] provides for explicit model checking, state space analysis and simulation. Symbolic model checking is provided for by the LTSmin toolset[2] [4,5].

Our work is part of INESS[3], an EC FP7-funded project, which aims at developing uniform specifications for future railway interlockings. Briefly, an interlocking prevents conflicting routes from being set by monitoring and controlling the operation of signals, points and other track side elements. Here, a "route" is the concept used by railway signallers to guide trains over arrangements of tracks, points etc. The high-level safety requirements of interlockings are to ensure that trains neither collide nor derail. In INESS, the functional requirements of interlockings are expressed as an xUML model, and one of the project tasks is to formally verify safety properties of such xUML interlocking models. Several approaches to this task are explored within the project. Our approach is based on model checking using mCRL2, and the current paper provides a first step by presenting a translation from xUML to mCRL2.

As in most real-world applications, the xUML models arising from the interlocking domain are of considerable size. One of our secondary aims in the current paper is therefore to investigate the feasibility of performing verification on the models resulting from translation. Moreover, we would like to explore the kind of behaviours contained in xUML interlocking models, and

Helle Hvid Hansen · Bas Luttik · MohammadReza Mousavi
Eindhoven University of Technology, Eindhoven, the Netherlands.
E-mail: {h.h.hansen,s.p.luttik,m.r.mousavi}@tue.nl

Jeroen Ketema · Jaco van de Pol
University of Twente, Enschede, the Netherlands.
E-mail: {j.ketema,j.c.vandepol}@ewi.utwente.nl

[1] `http://www.mcrl2.org`
[2] `http://fmt.cs.utwente.nl/tools/ltsmin/`
[3] `http://www.iness.eu`

expose any kind of semantic assumption that is either under-specified or beyond the scope of the model. To this end we also report on a small experiment using a toy interlocking specification, which was kindly provided to us by KnowGravity Inc.[4] This toy specification is almost as simple as it gets, but it shows, in particular, how different run-to-completion assumptions give rise to a wide variety of model sizes and observable traces.

The rest of this paper is organised as follows. In Section 2, we describe and motivate the use of model checking in the verification of xUML interlocking models. In Section 3, we introduce the xUML constructs covered by our translation, and discuss different types of run-to-completion assumptions. In Section 4, we describe our translation to mCRL2 and, in Section 5, we expose some issues related to model checking xUML models and report on our observations made in translating and verifying a toy xUML interlocking model. Finally, we discuss related work and conclude in Sections 6 and 7, respectively.

2 Model Checking xUML Interlocking Models

In model checking [2], a formal model describes all possible executions of the system being modelled, and verification is carried out by an exhaustive state space exploration of this model. In our case, we obtain a formal model as the mCRL2 translation of an instance of the xUML model. More precisely, a model is obtained by instantiating the classes and associations of the xUML interlocking model according to a particular track layout. A track layout is a configuration of physical and logical railway elements such as tracks, points, signals and routes (see Section 5.1 for an example). Model checking of xUML interlocking models is thus always carried out with respect to a particular instantiation of the model. Consequently, model checking cannot prove that all instances of the xUML model satisfy a given set of safety requirements.

In spite of the above limitation, model checking can provide valuable information: First, for a fixed track layout, verification is exhaustive, in contrast with simulation and testing. Secondly, the violation of a safety property in a particular model instance shows that the xUML model is not correct in general; traces that witness this undesired behaviour can be used to improve the model. Finally, we can increase confidence in the correctness of the xUML model by verifying particularly significant model instances. For example, ProRail[5], the Dutch railway infrastructure manager, has designed three track layouts that together are supposed to capture all features of track layouts found in the Netherlands. Proving that the model instances arising from these track layouts satisfy certain properties would increase the confidence that the properties also hold for other instances.

3 Executable UML

Executable UML (xUML) [15] consists of UML class diagrams, UML state machines and an action language which complies with the UML action semantics. There are several action languages in use; we refer to [15] for a — somewhat limited — overview.

The xUML models to be translated are expressed in the Cassandra/xUML dialect [14], as developed by KnowGravity. We briefly describe the modelling constructs relevant to us, following the UML 2.2 standard [17] where possible, and Cassandra/xUML otherwise.

3.1 Constructs

In class diagrams, we allow for class generalisations (inheritance) and associations between classes (specifying which class instantiations may reference each other). Association classes, however, may not occur, i.e., no objects may be related to instances of associations.

State machines may contain concurrent and composite states (AND- and OR-states) and initial pseudo-states. We currently do not translate history and final pseudo-states. All UML-defined transitions may occur as far as they involve the allowed (pseudo-)states. A transition is labelled with a trigger and a sequence of actions, both of which may be empty. A trigger must be a signal event or a change event.

Signals are communicated asynchronously. A signal can be sent either by an object within the system (an internal signal) or by the environment (an external signal). Each state machine is accompanied by an event pool which stores received signals until dispatched, i.e., until they are taken from the event pool by the state machine [17, Section 13.3.25].

A change event [17, Section 13.3.7] is an event which is generated when a certain condition becomes true. The condition typically refers to the states of objects referenced through associations. Contrary to the other modelling constructs relevant to us, the UML 2.2 semantics of change events [17, Section 13.3.7] is rather under-specified. For example, it is not detailed when

[4] `http://www.knowgravity.com`

[5] http://www.prorail.nl

a change event is evaluated or how a change event is detected. Also, implementations may or may not let change events remain in case their condition becomes false again after having been true. In Cassandra/xUML, change events are denoted by `when`(*cond*), where *cond* is a boolean expression. Given a state machine X with a transition t labelled by a change event `when`(*cond*), the Cassandra/xUML simulator adds an event $\mathtt{e}_{\mathtt{when}(cond)}$ to the event pool of X whenever *cond* changes from false to true (personal communication with KnowGravity). The event $\mathtt{e}_{\mathtt{when}(cond)}$ triggers transition t once dispatched and remains in the event pool even in case *cond* becomes false before $\mathtt{e}_{\mathtt{when}(cond)}$ is dispatched.

If a dispatched event is not the trigger of an enabled transition, the event is discarded. Otherwise, the actions labelling the transition are carried out. The only type of action we currently allow is the sending of a signal [17, Section 11.3.45], where the target may either be an object within the system or the environment.

3.2 Run-to-Completion

An important aspect of concurrency is the interleaving of process executions. Run-to-completion (RTC) assumptions can help reduce the complexity of a concurrent system. A *local RTC step* of a state machine X consists of processing all actions labelling a transition triggered by some event. In the literature, three different levels of RTC seem to be considered (no fixed terminology seems to exists and the names are our own):

Local RTC: A local RTC step of a state machine X must be completed before the next event can be dispatched to X.

Atomic RTC: While a state machine X is executing a local RTC step, no event can be dispatched to any of the state machines in the system.

Global RTC: External signals may only be dispatched to the system in case all event pools are empty and there are no remaining change events.

Local RTC is required by the UML specification [17, Section 15.3.12]. It ensures that a state machine is in a well-defined configuration before the next event is dispatched. With atomic RTC, local RTC steps in different state machines may not be interleaved (which is not forbidden by local RTC). Global RTC separates internal system interactions (between objects) from interactions with the environment. Note that atomic RTC implies local RTC, but that global RTC implies neither local nor atomic RTC. Atomic RTC is used in [20,9]. The Cassandra/xUML simulator employs both atomic RTC and global RTC [14, Section 4.3.5].

4 Translation into mCRL2

The mCRL2 specification language [12] extends the process algebra ACP [3] with abstract datatypes, including built-in types such as booleans, integers, and lists. New structured data types can be defined using the `struct` keyword. We currently use only enumerated data types, for example,

```
sort Elt_State = struct Ready | Not_Ready .
```

Functions over sorts can be defined by giving equations.

The process specification language of mCRL2 allows for the definition of basic actions with zero or more parameters. For example, `act send, read: Message` defines the actions `send` and `read` which take a parameter of type `Message`. Similarly, a process specification may take parameters, for example,

```
proc Element(state: Elt_State) .
```

Processes can be composed using sequential and parallel composition and non-deterministic choice. Moreover, actions can be hidden (turned into the silent action) and blocked (disallowed). Synchronisation rules take the form of so-called multi-actions $a_1 \mid a_2 \mid \ldots \mid a_n \rightarrow c$, which specify that the actions $a_1, a_2, \ldots, a_n$ must synchronise and the result of this multi-party synchronisation is the action c.

An mCRL2 specification consists of data type definitions, equations over the data types, process specifications and an initial process. The above process specification `proc Element(state: Elt_State)` could, for example, be initialised as `init Element(Ready)`.

4.1 Translation

In our translation from xUML to mCRL2, each class becomes a process specification. Each of these process specifications consists of two parallel parts: One part is the translation of the state machine associated with the class, the other part formalises the event pool associated with the state machine as a buffer process. The buffer process essentially implements a queue. An event is placed in the queue by a synchronous communication between the sending process and the buffer process. The sending process can be either the environment, another process representing a state machine, or a process monitoring change events (described at the end of this section). Signals are dispatched on a FIFO basis through synchronous communication between the buffer process and the process representing a state machine.

Class diagrams. As mentioned in Section 3.1, we allow for class generalisations and class associations. In our translation, the first is dealt with by flattening the class hierarchy; each superclass Y of a class X occurs only once in this flattening, even in case there are several *is-a* associations between X and Y in the class diagram. Now, if X is a class with superclasses $Y_1, \ldots, Y_n$, the flattened class X' arising from X has all attributes of $X, Y_1, \ldots, Y_n$, and the state machine of X' is defined as the concurrent composition of the state machines of $X, Y_1, \ldots, Y_n$.

Class associations are translated by defining an enumerated data type consisting of identifiers (depending on the instantiation of the model) and supplementing each mCRL2 representation of a class instance with one parameter (of the enumerated type) for each of its associations.[6] For example, if each instance of a class X is associated with exactly one instance of a class Y, then the process specification of X will be of the form `proc X(..., id_Y: Id, ...)`, where `Id` is the enumerated type consisting of identifiers.

State machines. The potential state configurations of a state machine X are encoded as follows. For each non-concurrent composite state (OR-state) S, we define an enumerated type `ancS_states` where `ancS` identifies S in the state hierarchy. If S has substates $P, \ldots, Q$, then `ancS_states` has members `ancS_substate_P`, ..., `ancS_substate_Q`, and `ancS_substate_nop`. The process specification of the state machine X is then supplied with a parameter `ancS_state` whose value represents the currently active substate of S. In particular, the top state T of a state machine for a class is always a non-concurrent state and gives rise to a parameter `T_state`. We refer to `ancS_state` as a *state parameter*. If S is not active, then `ancS_state` has the value `ancS_substate_nop`.

The configurations of a concurrent state S are not modelled by parameters, as they are determined by the state configurations of the (direct) substates of the concurrent components of S (the Cartesian product of these substates to be precise). To illustrate, consider the state machine F in Figure 1 (transitions are unlabelled as we only wish to illustrate how composite states are treated). In Figure 2 we list the data type definitions arising from F together with the declaration of state parameters in the process specification of F (disregarding any class attributes), and an initial process corresponding to the initial state configuration.

[6] We actually employ macro pre-processing of the mCRL2 specification before model checking. This is to avoid loop constructs when dealing with one-many and many-many associations.

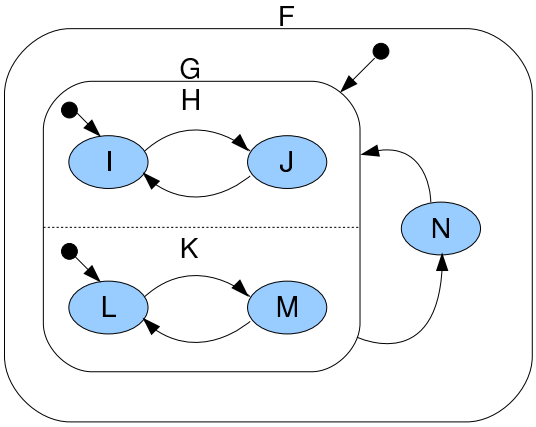

Fig. 1 A state machine

```
sort F_states     = struct F_substate_G
                         | F_substate_N
                         | F_substate_nop;
sort F_G_H_states = struct F_G_H_substate_I
                         | F_G_H_substate_J
                         | F_G_H_substate_nop;
sort F_G_K_states = struct F_G_K_substate_L
                         | F_G_K_substate_M
                         | F_G_K_substate_nop;

proc F(F_state: F_states,
       F_G_H_state : F_G_H_states,
       F_G_K_state : F_G_K_states) = ...;

init F(F_substate_G,
       F_G_H_substate_I,
       F_G_K_substate_L);
```

Fig. 2 Translation of the state machine in Figure 1: data types for representing states, state parameters and initialisation

Since we treat class generalisation by flattening, if a class Y generalises a class X, then the process specification of the state machine for X (which concurrently composes the state machines for Y and X) will have the state parameters arising from both X and Y. This is completely analogous to the handling of concurrent states within a state machine.

Transitions. Given the *local* RTC assumption, a process specifying the state machine of a class X can obtain an event from its buffer process (event pool) precisely when it is in a stable state, i.e., when no other transition is currently being taken. One of the transitions triggered by the obtained event is taken at random (non-deterministic choice), assuming such a transition exists. The actions labelling the chosen transition are executed and the state parameters of the process are updated to reflect the new state configuration. If no transition is triggered by the event, then the event is

discarded, as allowed by the UML state machine semantics [17, Section 15.3.12].

Change events. We implement change events by introducing a process for each such event. This process monitors the value of the condition in the change event. For example, if a state machine X has a transition from a state S triggered by (in pseudo-notation)

$\texttt{when}(P.\texttt{state} = T \;\&\; Q.\texttt{state} = U)$,

where P and Q are state machines associated with X, then we create a process

$\texttt{when\_X\_S}(\texttt{P\_in\_state\_T}\colon \texttt{Bool}, \texttt{Q\_in\_state\_U}\colon \texttt{Bool})$,

and let P and Q communicate synchronously with the monitor process whenever they enter or leave the states T and U, respectively. This communication updates the values of the boolean parameters `P_in_state_T` and `Q_in_state_U`. When an update results in the condition changing from false to true, the monitor process places a signal in the buffer of X. This signal remains in the buffer of X even when the condition becomes false again. Whence, at the moment the state machine reacts to the event, the condition might no longer hold.

Concretely, the monitor process `when_X_S` described above is specified as in Figure 3. The process defined in the figure can either receive an update from state machine P, represented by the action `when_upd_P`, or from Q, represented by the action `when_upd_Q`. We explain communication with P, the case of Q is analogous. The action `when_upd_P` carries a boolean data argument b which indicates whether P entered or left state T. The sum ensures that b can have either the value `True` or `False`. Upon reception of $\texttt{when\_upd\_P}(b)$, the process checks whether P was not in the state T before (`!P_in_State_T`), that P is in the state T now (b) and that Q is in state U (`Q_in_State_U`). If these conditions hold, a signal is put in the buffer of X (`send_when_to_X`) and the state of the monitor process is updated, written as $\texttt{when\_X\_S}(b,\ \texttt{Q\_in\_state\_U})$, after which the process is able to receive a state update again from either P or Q. If one of the conditions does not hold, the state of `when_X_S` is simply updated and again the process is able to receive a state update again from either P or Q.

Architecture. We now summarise how the elements of an xUML model are mapped onto the elements of an mCRL2 specification: For each flattened class with associated state machine X, we define a process specification $\texttt{proc X\_Complex}(\texttt{id}\colon \texttt{Id},\ \ldots)$ consisting of the parallel composition of $\texttt{X}(\texttt{id}\colon \texttt{Id},\ \ldots)$, $\texttt{X\_buffer}(\texttt{id}\colon \texttt{Id},\ \ldots)$, and $\texttt{X\_when}_i(\texttt{id}\colon \texttt{Id},\ \ldots)$ where i ranges over the change events of X. The parameter `id` represents the unique identifier associated with an instance of the represented class. The translation of the state machine X is embodied by `X`, and the process `X_buffer` represents the event pool. Each $\texttt{X\_when}_i$ monitors one of the change events occurring in the state machine associated with X, as described earlier.

```
proc when_X_S(P_in_state_T: Bool, Q_in_state_U: Bool) =
    ∑_{b: Bool} .when_upd_P(b).
    (
        (!P_in_State_T & b & Q_in_State_U) →
            send_when_to_X.when_X_S(b, Q_in_state_U)
        ◇
            when_X_S(b, Q_in_state_U)
    )
    +
    ∑_{b: Bool} .when_upd_Q(b).
    (
        (P_in_State_T &!Q_in_State_U & b) →
            send_when_to_X.when_X_S(P_in_state_U, b)
        ◇
            when_X_S(P_in_state_U, b)
    )
    ;
```

Fig. 3 A monitor process for a change event

An instance of an xUML model defines, in addition to the above, an enumerated type with object identifiers and an appropriate initialisation consisting of the parallel composition of the processes arising from the objects in the instantiation, together with the required synchronisation constraints.

5 Model Checking

From a model checking perspective the UML 2.2 semantics presents us with two problems:

1. Any instance of an xUML model has an infinite state space; this comes about, as UML 2.2 neither limits the size of event pools nor limits the number of events the instance may accept from the environment at any point in time.
2. Class instances may starve, i.e., a class instance may have events in its event pool waiting to be dispatched, but the instance may never get its turn; this comes about, as UML 2.2 does not impose any fairness restrictions concerning event dispatching.

Since our translation to mCRL2 faithfully follows the UML 2.2 semantics, the obtained mCRL2 models will also have both infinite state spaces and leave room for starvation. To alleviate these problems, we propose two constraints on our mCRL2 models:

A. Limit the size of buffers. This restriction will overcome the first of the above problems. However, as

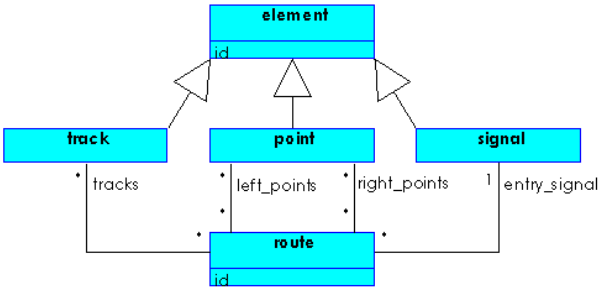

Fig. 4 Class diagram for Micro interlocking

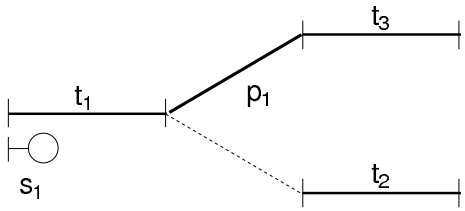

Fig. 5 Track layout for an instance of the Micro model

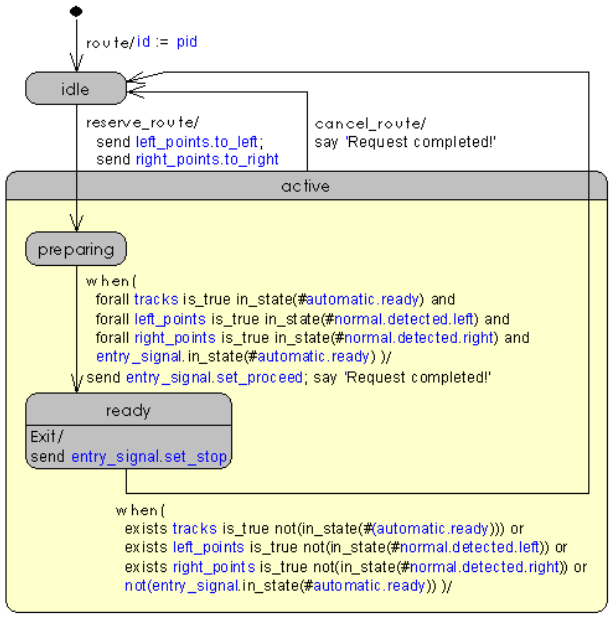

Fig. 6 State machine for class Route

an object may send several signals to itself during a local RTC step (cf. Section 3.2), we only impose this limit with regard to messages coming from other objects and from the environment in order to avoid deadlock.

B. Add a mechanism which ensures that the system can only receive a message from the environment in case all message queues are empty. In other words, implement global RTC (cf. Section 3.2). This restriction addresses both of the above problems under the assumption that external signals do not (directly or indirectly) trigger an unbounded number of internal events. Consequently, each process will eventually get the chance to empty its buffer.

It should be clear that both A and B only eliminate traces from the translated model. Hence, as safety properties are violated by finite traces, any safety violation found in a model restricted according to A or B is also present in the unrestricted model.

5.1 Model Checking a Toy Specification

We have applied the translation from Section 4 to a toy interlocking specification which we refer to as the Micro model. The Micro model has classes named *element*, *track*, *point*, *signal* and *route*, where element is a generalisation of track, point and signal. The class diagram for this model is depicted in Figure 4. An instance of the Micro model is obtained from the track layout depicted in Figure 5.

The layout consists of three tracks t_1, t_2, t_3, one point p_1 (positioned left in the picture), one signal s_1, and two routes: route r_1 requires p_1 to be positioned left and goes from track t_1 to track t_3; route r_2 requires p_1 to be positioned right and goes from t_1 to t_2; both routes have s_1 as their entry signal. The model instance thus contains three track objects, one point object, one signal object and two route objects, and these objects are linked via the following associations:

$$\begin{aligned} \mathit{tracks} &= \{\langle r_1, t_1\rangle, \langle r_1, t_3\rangle, \langle r_2, t_1\rangle, \langle r_2, t_2\rangle\} \\ \mathit{left\_points} &= \{\langle r_1, p_1\rangle\} \\ \mathit{right\_points} &= \{\langle r_2, p_1\rangle\} \\ \mathit{entry\_signal} &= \{\langle r_1, s_1\rangle, \langle r_2, s_1\rangle\} \end{aligned}$$

The main functionality of the Micro model is route setting and route cancellation. Informally described, when a route receives a reserve request, it should signal to its left points and right points to move into position. When all points are positioned, all tracks along the route are clear and the entry signal is ready, the entry signal is set to show proceed. When one of the elements associated with the route is no longer in the required state, or the route is cancelled, the route entry signal is set to show stop. The state machine describing the behaviour of the class Route is shown in Figure 6.

We translated the above instance of the Micro model into mCRL2 in two different ways corresponding to constraints A and B from the previous section. The state space resulting from version A is huge (61×10^{12} states) even with buffer size 1, but our symbolic tool LTSmin still computes the number of states within seconds: The mentioned state space was explored in 113 seconds using 238 MB memory of a machine equipped with an Intel Xeon 2.66 GHz, 32GB of memory and Linux 2.6.18.

To obtain version B we used barrier synchronisation. This version has a significantly smaller state space with 8 million states. However, computing the number of states takes longer and uses more memory (160 seconds and 311MB). The state space reduction that stems from a global RTC assumption is thus significant, but runtime increases for the symbolic tools.

We were able to prove the presence of certain (unwanted) traces in both version A and B by placing a monitor process in parallel with the mCRL2 translation of Micro. The monitor deadlocks the process in case a certain finite trace, representing the violation of a safety property, is present. The deadlock detection functionality of LTSmin was used to produce a trace. The trace shows that when the entry signal s_1 has been set to show proceed, the system may command the point p_1 to move *before* it sets s_1 to show stop, thus risking the derailment of a train passing over p_1. However, we point out that the Micro model is merely intended to illustrate the type of xUML model constructs that are used in the modelling of interlockings, and it is not claimed to be a safe interlocking specification. Our main point is that some of these traces cannot be observed in the Cassandra/xUML simulator, because it uses a stronger RTC assumption (atomic and global RTC) than our versions A (local RTC) and B (local and global RTC).

6 Related Work

There is extensive work on the formalisation of executable UML, and in particular, UML state machines, for the purpose of carrying out formal verification [1, 20,7,9,13]. Translation of xUML into a process algebraic language occurs in [19,22]. More references can be found in [7] and in the survey article [18].

In all aforementioned work, translation focuses on composite and concurrent states, history pseudo-states, (conflicting) transitions, and the action language. We were not able to locate any research that includes the formalisation of change events. The reason for this could be that change events are considered a problematic construct due to their under-specified semantics. An indication hereof is given by the fact that the "foundational subset for xUML" (fUML) [16] forbids the use of change events. Nonetheless, we want to include change events, as they are an essential ingredient of the interlocking specifications provided to us. Consequently, we cannot (completely) base our work on the formally better specified fUML.

Formal methods have been widely applied in the verification of interlockings. The work can be divided into two categories: Verification of concrete interlockings (see, e.g., [11,6,10]) and verification of more high-level interlocking specifications (see, e.g., [21,8]), as in our case.

7 Conclusion

We have presented a translation of a subset of xUML into the process algebra mCRL2. Each of the elements of the xUML subset is translated into a very simple mCRL2 construct, with the translation of the change events being the most complicated. Given the simplicity of the mCRL2 constructs, we expect that our translation can be automated without too much trouble. In fact, work on this automatic translation from xUML (in the form of XMI files) to mCRL2 has already begun.

Future work includes extending our translation to further constructs that occur in larger xUML interlocking models provided to us. These constructs include transition guards and call events (synchronous communication), which we expect are easily added. In fact, we expect that our translation can also easily be extended to include history and final pseudo-states, and even any chosen action language, bar operations like object creation and destruction. These last operations would correspond to on-the-fly process creation and destruction which are not possible in mCRL2. Other future work includes defining a formal semantics of xUML interlocking models, and proving the correctness of our translation with respect to this semantics.

Our first steps towards verifying safety properties of xUML interlocking specifications have demonstrated the following:

- The fairness and safety properties of an interlocking system may depend crucially on the run-to-completion (RTC) assumption employed in the implementation. Verification should therefore be relative to a particular choice of RTC.
- Even for small xUML models, such as our toy specification, the state space can be enormous. Still the symbolic model checker seems to be able to deal quite well with the mCRL2 translations obtained from the toy specification. However, in order to verify real interlocking specifications, we expect it will be necessary to come up with specific abstraction and decomposition techniques, as well as reduce the number of event orderings, either in a generic way (partial-order reduction) or specifically (by using different RTC assumptions).

Funding. This research is partially funded by the European Comission (EC), as a grant to the FP7 project INESS, grant agreement no. 218575. Any opinions, findings and conclusions or recommendations expressed in

this material are those of the authors and do not necessarily reflect the views of either the EC or the INESS consortium.

References

1. R. Alur and M. Yannakakis. Model checking of hierarchical state machines. *ACM Transactions on Programming Languages and Systems*, 23(3):273–303, 2001.
2. C. Baier and J.-P. Katoen. *Principles of Model Checking*. The MIT Press, 2008.
3. J. A. Bergstra and J. W. Klop. Process algebra for synchronous communication. *Information and Control*, 60(1-3):109–137, 1984.
4. S. Blom and J. van de Pol. Symbolic reachability for process algebras with recursive data types. In *Proc. Theoretical Aspects of Computing (ICTAC 2008)*, volume 5160 of *Lecture Notes in Computer Science*, pages 81–95. Springer, 2008.
5. S. C. C. Blom, J. C. van de Pol, and M. Weber. Bridging the gap between enumerative and symbolic model checkers. Technical Report TR-CTIT-09-30, CTIT, University of Twente, Enschede, 2009.
6. A. Cimatti, F. Giunchiglia, G. Mongardi, D. Romano, F. Torielli, and P. Traverso. Formal verification of a railway interlocking system using model checking. *Formal Aspects of Computing*, 10(4):361–380, 1998.
7. W. Damm, B. Josko, A. Pnueli, and A. Votintseva. A discrete-time UML semantics for concurrency and communication in safety-critical applications. *Science of Computer Programming*, 55:81–155, 2005.
8. L.-H. Eriksson. Specifying railway interlocking requirements for practical use. In *Proceedings of the 15th International Conference on Computer Safety, Reliability and Security (SAFECOMP'96)*. Springer, 1996.
9. Fei Xie, V. Levin, and J. Browne. Model checking for an executable subset of UML. In *16th IEEE International Conference on Automated Software Engineering (ASE 2001)*, pages 333–336, 2001.
10. W. Fokkink. Safety criteria for the vital processor interlocking at Hoorn-Kersenboogerd. In *5th Conference on Computers in Railways (COMPRAIL'96). Volume I: Railway Systems and Management*, 1996.
11. S. Gnesi, D. Latella, G. Lenzini, C. Abbaneo, A. M. Amendola, and P. Marmo. An automatic SPIN validation of a safety critical railway control system. In *Proc. of the 2000 Int. Conf. on Dependable Systems and Networks*, pages 119–124, Washington, DC, USA, 2000. IEEE Computer Society.
12. J. F. Groote, A. Mathijssen, M. A. Reniers, Y. S. Usenko, and M. van Weerdenburg. The formal specification language mCRL2. In *Proc. of Methods for Modelling Software Systems,*, volume 06351 of *Dagstuhl Seminar Proceedings*, 2007.
13. Z. Hu and S. M. Shatz. Explicit modeling of semantics associated with composite states in UML statecharts. *Journal of Automated Software Engineering*, 13(4):423–467, Oct. 2006.
14. KnowGravity. *Cassandra/xUML User's Guide*, 2008.
15. S. J. Mellor and M. Balcer. *Executable UML: A foundation for model-driven architecture*. Addison Wesley, 2002.
16. Object Management Group. Semantics of a Foundational Subset for Executable UML Models, Nov. 2008.
17. Object Management Group. OMG Unified Modeling Language Superstructure Version 2.2, Feb. 2009.
18. B. Purandar and S. Ramesh. Model checking of statechart models: Survey and research directions, July 2004.
19. E. Turner, H. Treharne, S. Schneider, and N. Evans. Automatic generation of CSP || B skeletons from xUML models. In *Proc. of Theoretical Aspects of Computing (ICTAC 2008)*, pages 364–379, 2008.
20. M. von der Beeck. Formalization of UML-statecharts. In *Proc. UML 2001*, volume 2185 of *Lecture Notes in Computer Science*, pages 406–421. Springer, 2001.
21. K. Winter and N. J. Robinson. Modelling large railway interlockings and model checking small ones. In *ACSC '03: Proc. of the 26th Australasian comp. sci. conference*, pages 309–316. Australian Computer Society, Inc., 2003.
22. W. L. Yeung, K. R. P. H. Leung, J. Wang, and W. Dong. Improvements towards formalizing UML state diagrams in CSP. In *Proc. of the 12th Asia-Pacific Software Engineering Conference (APSEC 2005)*. IEEE Computer Society, 2005.